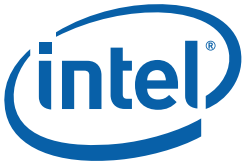
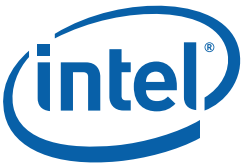


Intel® Math Kernel Library

User's Guide for Linux*

Document Number: 314774-001US

World Wide Web:
http://www.intel.com/cd/software/products/asmo-na/eng/perflib/mkl/index.htm



Disclaimer and Legal Information

Verision Information

Version	Version Information	Date
-001	Original issue.	09-2006

Contents

Tables

1 Overview

Intel® Math Kernel Library (Intel® MKL) offers highly optimized, thread-safe math routines for science, engineering, and financial applications that require maximum performance.

About This Document

To get Intel® MKL reference information, see *Intel MKL Reference Manual*, which features routines functionality, parameters description, interfaces and calling syntax as well as return values. However, a lot of questions not answered in the Reference Manual arise when you try to call Intel MKL routines from your applications. For example, you need to know how the library is organized, how to configure Intel MKL for your particular platform and problems you are solving, how to compile and link your applications with Intel MKL. You also need understanding of how to achieve best performance, take advantage of Intel MKL threading and memory management. Other questions may deal with specifics of routine calls, for example, passing parameters in different programming languages or coding inter-language routine calls. You may be interested in the ways of estimating and improving computation accuracy. These and similar issues make up Intel MKL *usage information*.

Purpose

This document focuses on the usage information needed to call Intel MKL routines from user's applications running on Linux*. Linux usage of Intel MKL has its particular features, which are described in this guide, along with those that do not depend upon a particular OS.

This guide contains usage information for plain, non-cluster, Intel MKL routines and functions, comprised in the function domains listed in Table 8 (in Appendix A). Usage information inherent to functions and routines available only with Intel MKL Cluster Edition will be included in the appropriate user's guide to be developed.

Audience

The guide is intended for Linux programmers whose software development experience may vary from beginner to advanced.

Notational Conventions

The document employs the following font conventions and symbols:

Italic	Italic is used for emphasis and also indicates document names in body text, for example, see *Intel MKL Reference Manual*
`Monospace lowercase`	Indicates filenames, directory names and pathnames, for example `libmkl_ia32.a ,/opt/intel/mkl/9.0 ;`
`Monospace lowercase mixed with uppercase`	Indicates commands and command-line options, for example, `ifort myprog.f -L$MKLPATH -lmkl_blas95 ;` C/C++ code fragments, for example, `complex16 a[N], b[N], c;`
`UPPERCASE MONOSPACE`	System variables, for example, `$MKLPATH`
`Monospace italic`	Indicates a parameter of a makefile or in other contexts, for example, *`functions_list`* When enclosed in angle brackets, indicates a placeholder for an identifier, an expression, a string, a symbol, or a value. Substitute one of these items for the placeholder, for example, *`<mkl directory>`*
[items]	Square brackets indicate that the items enclosed in brackets are optional.
{ item | item }	Braces indicate that only one of the items listed between braces should be selected. A vertical bar (|) separates the items.

2 Before You Begin

Before you get started using the Intel® Math Kernel Library (Intel® MKL), sorting out a few important basic concepts will greatly help you get off to a good start. The table below summarizes some important things to think of before you start using Intel MKL.

Table 1 What you need to know before you get started

Target platform	Identify the processor inside your target machine: • IA-32 • Processor with Intel® EM64T • Intel® Itanium® processor family. **Reason.** When linking your application with the Intel MKL libraries, the directory corresponding to your particular architecture should be included in the link command (see Selecting libraries to link for your platform and function domain).
Mathematical problem	Identify all Intel MKL function domains that problems you are solving require: • BLAS • Sparse BLAS • LAPACK • Sparse Solver routines • Vector Mathematical Library functions • Vector Statistical Library functions • Fourier Transform functions • Interval Solver routines • Trigonometric Transform routines • Fast Poisson, Laplace, and Helmholz Solver routines. **Reason.** The function domain you intend to use determines the MKL libraries that your application must link with. For more information see Selecting libraries to link for your platform and functional domain.
Programming language	Though Intel MKL provides support for both Fortran and C/C++ programming, not all the function domains support a particular language environment, for example, C/C++ or Fortran90/95. Identify the languages that your function domains support (see Appendix A: Intel MKL Language Support). **Reason.** In case your function domain does not directly support the needed environment, you can use mixed-language programming. See Calling LAPACK, BLAS, and CBLAS routines from C language environments. See also Using language-specific interfaces with Intel MKL for a list of language-

	specific interface libraries and modules and an example how to generate them.
Threading model	Select among the following options how you are going to thread your application: • Your application is already threaded • You want to use MKL treading capability (the `libguide` library) • You do not want to thread your application **Reason.** By default Intel MKL runs with a number of threads equal to one, except for Direct Sparse Solver. To utilize multi-threading, you will need to set the number of threads yourself. For more information, and especially, how to avoid conflicts in the threaded execution environment, see Using Intel MKL parallelism.
Linking model	Decide which linking model is appropriate for linking you application with Intel MKL libraries: • Static • Dynamic **Reason.** For information on the benefits of each linking model, link command syntax and examples, link libraries as well as on other linking topics, like how to save disk space by creating a custom dynamic library, see Linking Your Application with Intel® Math Kernel Library.

3 Linking Your Application with Intel® Math Kernel Library

This chapter features linking of your applications with Intel® Math Kernel Library (Intel® MKL). The chapter discusses the library structure, as it determines much of the linking procedure; compares static and dynamic linking models; describes the general link line syntax to be used for linking with Intel MKL libraries and finally provides comprehensive information in a tabular form on the libraries that should be linked with your application for a particular platform and function domain. Generation of language-specific interface libraries and modules as well as building of custom dynamic libraries is also discussed.

3.1 Intel MKL Library Structure

The table below shows a high-level structure for Intel MKL after installation.

Table 2 High-Level directory structure

Directory	Comment
<mkl directory>	Main directory, for example, "/opt/intel/mkl/9.0"
<mkl directory>/doc	Documentation directory
<mkl directory>/man/man3	Man pages for Intel MKL BLAS, Sparse BLAS, and LAPACK (without auxiliary) functions
<mkl directory>/examples	Source and data for examples
<mkl directory>/include	Contains INCLUDE files for both library routines and test and example programs
<mkl directory>/interfaces/blas95	Contains f95 wrappers for BLAS and makefile to build the library
<mkl directory>/interfaces/lapack95	Contains f95 wrappers for LAPACK and makefile to build the library
<mkl directory>/interfaces/fftw2xc	Contains wrappers for FFTW version 2.x (C interface) to call Intel MKL DFTI
<mkl directory>/interfaces/fftw2xf	Contains wrappers for FFTW version 2.x (Fortran interface) to call Intel MKL DFTI
<mkl directory>/interfaces/fftw3xc	Contains wrappers for FFTW version 3.x (C interface) to call Intel MKL DFTI

Directory	Comment
<*mkl directory*>/interfaces/fftw3xf	Contains wrappers for FFTW version 3.x (Fortran interface) to call Intel MKL DFTI
<*mkl directory*>/interfaces/fftc	Contains wrappers for FFT (C interface) to call Intel MKL DFTI
<*mkl directory*>/interfaces/fftf	Contains wrappers for FFT (Fortran interface) to call Intel MKL DFTI
<*mkl directory*>/tests	Source and data for tests
<*mkl directory*>/lib/32	Contains static libraries and shared objects for IA-32 applications
<*mkl directory*>/lib/em64t	Contains static libraries and shared objects for applications running on processors with Intel® EM64T
<*mkl directory*>/lib/64	Contains static libraries and shared objects for Intel® Itanium®2 processor
<*mkl directory*>/tools/builder	Contains tools for creating custom dynamically linkable libraries
<*mkl directory*>/tools/environment	Contains shell scripts to set environmental variables in the user shell
<*mkl directory*>/tools/support	Contains a utility for reporting the package ID and license key information to Intel® Premier Support
<*mkl directory*>/tools/plugins/com.intel.mkl.help	Contains an Eclipse plugin with Intel MKL Reference Manual in WebHelp format.

Intel® MKL separates IA-32 library versions, Intel® EM64T versions, and versions for Intel® Itanium® 2 processor:

- The IA-32 versions are located in the `lib/32` directory.
- Intel® EM64T versions are located in the `lib/em64t` directory.
- Intel® Itanium® 2 processor versions are located in the `lib/64` directory.

See detailed structure of these directories in Table 3.

Intel® MKL consists of two parts:

- high-level libraries (LAPACK, sparse solver)
- processor-specific kernels in `libmkl_ia32.a`, `libmkl_em64t.a`, and `libmkl_ipf.a`.

High-level libraries

The high-level libraries are optimized without regard to the processor and can be used effectively on processors from Intel® Pentium® processor through Intel® Core™ 2 Extreme processor family and Intel® Itanium® 2 processor.

Processor-specific kernels

Processor-specific kernels containing BLAS, Sparse BLAS, CBLAS, GMP, FFTs, DFTs, VSL, VML, interval arithmetic, Trigonometric Transform and Poisson Library routines are optimized for each specific processor.

Threading libraries

Threading software is supplied in two versions:

- separate library (`libguide.a`) for static linking (discouraged - please see Intel MKL-specific linking recommendations)
- dynamic link library (`libguide.so`) for linking dynamically to Intel® MKL (recommended - please see Intel MKL-specific linking recommendations).

Directory structure in detail

The information in the table below shows detailed structure of the architecture-specific directories of the library. For the detailed structure of the `doc` directory, see Appendix B.

Table 3 Detailed Intel® MKL directory structure

Directory/file		Contents
`lib/32`		Contains all libraries for 32-bit applications
	`libmkl_ia32.a`	Optimized kernels (BLAS, CBLAS, Sparse BLAS, GMP, FFTs, DFTs, VML, VSL, interval arithmetic) for 32-bit applications
	`libmkl_lapack.a`	LAPACK routines and drivers
	`libmkl_solver.a`	Sparse solver routines
	`libguide.a`	Threading library for static linking
	`libmkl.so`	Library dispatcher for dynamic load of processor specific kernel
	`libmkl_lapack32.so`	LAPACK routines and drivers, single precision data types
	`libmkl_lapack64.so`	LAPACK routines and drivers, double precision data types
	`libmkl_def.so`	Default kernel (Intel® Pentium®, Pentium® Pro, and Pentium® II processors)

	libmkl_p3.so	Intel® Pentium® III processor kernel
	libmkl_p4.so	Pentium® 4 processor kernel
	libmkl_p4p.so	Kernel for Intel® Pentium® 4 processor with Streaming SIMD Extensions 3 (SSE3)
	libmkl_p4m.so	Kernel for processors based on the Intel® Core™ microarchitecture
	libvml.so	Library dispatcher for dynamic load of processor specific VML kernels
	libmkl_vml_def.so	VML part of default kernel (Pentium®, Pentium® Pro, Pentium® II processors)
	libmkl_vml_p3.so	VML part of Pentium® III processor kernel
	libmkl_vml_p4.so	VML part of Pentium® 4 processor kernel
	libmkl_vml_p4p.so	VML for Pentium® 4 processor with Streaming SIMD Extensions 3 (SSE3)
	libmkl_vml_p4m.so	VML for processors based on the Intel® Core™ microarchitecture
	libmkl_ias.so	Interval arithmetic routines
	libguide.so	Threading library for dynamic linking
lib/em64t		Contains all libraries for Intel® EM64T applications
	libmkl_em64t.a	Optimized kernels for Intel® EM64T
	libmkl_lapack.a	LAPACK routines and drivers
	libmkl_solver.a	Sparse solver routines
	libguide.a	Threading library for static linking
	libmkl.so	Library dispatcher for dynamic load of processor specific kernel
	libmkl_lapack32.so	LAPACK routines and drivers, single precision data types
	libmkl_lapack64.so	LAPACK routines and drivers, double precision data types
	libmkl_def.so	Default kernel
	libmkl_p4n.so	Kernel for Intel® Xeon® processor with Intel® EM64T
	libmkl_mc.so	Kernel for processors based on the Intel® Core™ microarchitecture
	libvml.so	Library dispatcher for dynamic load of processor specific VML kernels
	libmkl_vml_def.so	VML part of default kernel
	libmkl_vml_p4n.so	VML for Intel® Xeon® processor with Intel® EM64T
	libmkl_vml_mc.so	VML for processors based on the Intel® Core™ microarchitecture
	libmkl_ias.so	Interval arithmetic routines
	libguide.so	Threading library for dynamic linking
lib/64		Contains all libraries for Intel® Itanium® 2-based applications
	libmkl_ipf.a	Processor kernels for Intel® Itanium® 2 processor

	libmkl_lapack.a	LAPACK routines and drivers
	libmkl_solver.a	Sparse solver routines
	libguide.a	Threading library for static linking
	libmkl_lapack32.so	LAPACK routines and drivers, single precision data types
	libmkl_lapack64.so	LAPACK routines and drivers, double precision data types
	libguide.so	Threading library for dynamic linking
	libmkl.so	Library dispatcher for dynamic load of processor specific kernel
	libmkl_i2p.so	Itanium® 2 processor kernel
	libmkl_vml_i2p.so	Itanium® 2 processor VML kernel
	libmkl_ias.so	Interval arithmetic routines
	libvml.so	Library dispatcher for dynamic load of processor specific VML kernel

Additionally, a number of interface libraries may be generated (see Using language-specific interfaces with Intel MKL).

3.2 Selecting Between Static and Dynamic Linking

You can link your applications with Intel MKL libraries statically, using static library versions, or dynamically, using shared libraries.

Static linking

During static linking, all links are resolved at link time. Therefore, the behavior of statically built executables is absolutely predictable, as they do not depend upon a particular version of the libraries available on the system where the executables run. Such executables must behave exactly the same way as was observed during testing. The main disadvantage of static linking is that upgrading statically linked applications to higher library versions is troublesome and time-consuming, as you have to relink the entire application. Besides, static linking produces large-size executables and uses memory inefficiently, since if several executables are linked with the same library, each of them loads it into memory independently. However, this is hardly an issue for Intel MKL, used mainly for large-size problems. — It matters only on shared-memory systems for executables having data size relatively small and comparable with the size of the executable.

Dynamic linking

During dynamic linking, resolving of some undefined symbols is postponed until run time. Dynamically built executables still contain undefined symbols along with lists of

libraries that provide definitions of the symbols. When the executable is loaded, final linking is done before the application starts running. If several dynamically built executables use the same library, the library loads to memory only once and the executables share it, thereby saving memory. Dynamic linking ensures consistency in using and upgrading libraries, as all the dynamically built applications share the same library. This way of linking enables you to separately update libraries and applications that use the libraries, which facilitates keeping applications up-to-date. The advantages of dynamic linking are achieved at the cost of run-time performance losses, as a part of linking is done at run time and every unresolved symbol has to be looked up in a dedicated table and resolved.

Making the choice

It is up to you to select whether to link in Intel MKL libraries dynamically or statically when building your application.

In most cases, users choose dynamic linking due to its strong advantages.

However, if you are developing applications to be shipped to a third-party, to have nothing else than your application shipped, you have to use static linking.

Intel MKL-specific linking recommendations

You are strongly encouraged to dynamically link in Intel MKL threading library `libguide`. Linking to static OpenMP run-time libraries is not recommended, as it is very easy with layered software to link in more than one copy of them. This causes performance problems (too many threads) and may also cause correctness problems if more than one copy is initialized (For more information, see Using Intel MKL parallelism).

3.3 Link Command Syntax

To link libraries with names `libxxx.a` or `libxxx.so` with your application, two options are available:

- Just list library names using relative or absolute paths, for example,

  ```
  <ld> myprog.o /opt/intel/cmkl/9.0/lib/32/libmkl_scalapack.a
  /opt/intel/cmkl/9.0/lib/32/libmkl_blacs.a
  /opt/intel/cmkl/9.0/lib/32/libmkl_lapack.a
  /opt/intel/cmkl/9.0/lib/32/libmkl_ia32.a
  /opt/intel/cmkl/9.0/lib/32/libguide.so -lpthread
  ```

  where `<ld>` is a linker, `myprog.o` is a user's object file, then Intel MKL libraries are listed followed by the system library `libpthread`

- In the link line, list library names (with absolute or relative paths, if needed, preceded with `-L <path>`, which indicates where to search for binaries. Discussion of linking with Intel MKL libraries employs this option.

To link with Intel MKL libraries, follow this general form of specifying the path and libraries in the link line:

```
-L<MKL path>
[-lmkl_solver] [-lmkl_lapack95] [-lmkl_blas95]
{[-lmkl_lapack] -lmkl_{ia32, em64t, ipf},[-lmkl_lapack{32,64}]
-lmkl, -lvml}
-lguide –lpthread [-lm]
```

NOTE: It is necessary to follow the order of listing libraries in the link command because Linux does not support multi-pass linking.

See Selecting libraries to link for your platform and functional domain for specific recommendations on which libraries to link in depending on your Intel MKL usage scenario.

3.4 Selecting Libraries to Link for Your Platform and Function Domain

Using the link command (see Link command syntax and some examples), note that

- `libmkl_solver.a` contains the sparse solver functions,
- `libmkl_lapack.a`, or `libmkl_lapack32.so` and `lib_mkl_lapack64.so` have the LAPACK functions.
- `libmkl_ia32.a, libmkl_em64t.a`, and `libmkl_ipf.a` have the BLAS, Sparse BLAS, GMP, FFT/DFT, VML, VSL, interval arithmetic, Trigonometric Transform and Poisson Library functions for IA-32, Intel® EM64T, and Intel® Itanium® processors respectively.
  `libmkl_lapack95.a` and `libmkl_blas95.a` contain LAPACK95 and BLAS95 interfaces respectively. They are not included into the original distribution and should be built before using the interface (see Fortran-95 interfaces and wrappers to LAPACK and BLAS for details on building the libraries).
  The `libmkl.so` file contains the dynamically loaded versions of these objects except for VML/VSL, which are contained in `libvml.so`.
  In all cases, appropriate libraries will be loaded at runtime.
- The `-lguide` option is used to link in the threading library libguide (see Intel MKL-specific linking recommendations).
- If you want to use FFT/DFT, you may link in the Linux mathematics library libm by adding "`-lm`".
- In products for Linux*, it is necessary to link to the `pthreads` library by adding `-lpthread`. The `pthread` library is native to Linux* and Intel® MKL makes use of this library to support multi-threading. Depending on what functions you call, you

may have to include this at the end of your link line (link order is important) regardless of whether you plan to use more than one thread.

Table 4, Table 5, and Table 6 illustrate the choice of libraries for dynamic and static linking and different architectures. To link in a library `libxxx`, you should include "`-lxxx`" into the link command.

Table 4 Link libraries for IA-32 applications by function domain

Function domain / Interface	Intel MKL libraries in lib/em64t directory		Linux libraries	
	Dynamic		Dynamic	Static
BLAS	libmkl.so libguide.so	libmkl_ia32.a libguide.a[1]	libpthread.so	libpthread.a
Sparse BLAS	libmkl.so libguide.so	libmkl_ia32.a libguide.a	libpthread.so	libpthread.a
BLAS95 Interface	libmkl_blas95.a libmkl.so libguide.so	libmkl_blas95.a libmkl_ia32.a libguide.a	libpthread.so	libpthread.a
CBLAS	libmkl.so libguide.so	libmkl_ia32.a libguide.a	libpthread.so	libpthread.a
LAPACK	libmkl_lapack32.so and/or libmkl_lapack64.so libmkl.so libguide.so	libmkl_lapack.a libmkl_ia32.a libguide.a	libpthread.so	libpthread.a
LAPACK95 Interface	n/a[2]	libmkl_lapack95.a libmkl_lapack.a libmkl_ia32.a libguide.a	libpthread.so	libpthread.a
Sparse Solver	n/a	libmkl_solver.a libmkl_lapack.a libmkl_ia32.a libguide.a	libpthread.so	libpthread.a
Vector Math Library	libvml.so libguide.so	libmkl_ia32.a libguide.a	libpthread.so libm.so	libpthread.a libm.a

Function domain / Interface	Intel MKL libraries in lib/em64t directory		Linux libraries	
	Dynamic		Dynamic	Static
Vector Statistical Library	libmkl_lapack32.so and/or libmkl_lapack64.so libmkl.so libvsl.so libguide.so	libmkl_lapack.a libmkl_ia32.a libguide.a	libpthread.so libm.so	libpthread.a libm.a
Fourier Transform Functions	libmkl.so libguide.so	libmkl_ia32.a libguide.a	libpthread.so libm.so (optionally)	libpthread.a libm.a (optionally)
Interval Arithmetic	libmkl_lapack32.so and/or libmkl_lapack64.so libmkl.so libguide.so	libmkl_lapack.a libmkl_ia32.a libguide.a	libpthread.so	libpthread.a
Trigonometric Transform Functions	libmkl.so libvml.so libguide.so	libmkl_ia32.a libguide.a	libpthread.so	libpthread.a
Poisson Library	libmkl.so libvml.so libguide.so	libmkl_ia32.a libguide.a	libpthread.so	libpthread.a

[1] Regardless of the function domain, when linking statically with `libguide` (discouraged), sometimes, you may have to use the libguide version different from the one in the indicated directory (see Notes below).

[2] Not applicable

Table 5 Link libraries for Intel® EM64T applications by function domain

Function domain / Interface	Intel MKL libraries in lib/em64t directory		Linux libraries	
	Dynamic	Static	Dynamic	Static
BLAS	libmkl.so libguide.so	libmkl_em64t.a libguide.a[1]	libpthread.so	libpthread.a
Sparse BLAS	libmkl.so libguide.so	libmkl_em64t.a libguide.a	libpthread.so	libpthread.a
BLAS95 Interface	libmkl_blas95.a libmkl.so libguide.so	libmkl_blas95.a libmkl_em64t.a libguide.a	libpthread.so	libpthread.a
CBLAS	libmkl.so libguide.so	libmkl_em64t.a libguide.a	libpthread.so	libpthread.a
LAPACK	libmkl_lapack32.so and/or libmkl_lapack64.so libmkl.so libguide.so	libmkl_lapack.a libmkl_em64t.a libguide.a	libpthread.so	libpthread.a
LAPACK95 Interface	n/a[2]	libmkl_lapack95.a libmkl_lapack.a libmkl_em64t.a libguide.a	libpthread.so	libpthread.a
Sparse Solver	n/a	libmkl_solver.a libmkl_lapack.a libmkl_em64t.a libguide.a	libpthread.so	libpthread.a
Vector Math Library	libvml.so libguide.so	libmkl_em64t.a libguide.a	libpthread.so libm.so	libpthread.a libm.a
Vector Statistical Library	libmkl_lapack32.so and/or libmkl_lapack64.so libmkl.so libvsl.so libguide.so	libmkl_lapack.a libmkl_em64t.a libguide.a	libpthread.so libm.so	libpthread.a libm.a

Function domain / Interface	Intel MKL libraries in lib/em64t directory		Linux libraries	
	Dynamic	Static	Dynamic	Static
Fourier Transform Functions	libmkl.so libguide.so	libmkl_em64t.a libguide.a	libpthread.so libm.so (optionally)	libpthread.a libm.a (optionally)
Interval Arithmetic	libmkl_lapack32.so and/or libmkl_lapack64.so libmkl.so libguide.so	libmkl_lapack.a libmkl_em64t.a libguide.a	libpthread.so	libpthread.a
Trigonometric Transform Functions	libmkl.so libvml.so libguide.so	libmkl_em64t.a libguide.a	libpthread.so	libpthread.a
Poisson Library	libmkl.so libvml.so libguide.so	libmkl_em64t.a libguide.a	libpthread.so	libpthread.a

[1] Regardless of the function domain, when linking statically with `libguide` (discouraged), sometimes, you may have to use the libguide version different from the one in the indicated directory (see Notes below).

[2] Not applicable

Table 6 Link libraries for Itanium® 2-based applications by function domain

Function domain/ Interface	Intel MKL libraries in lib/64 directory		Linux libraries	
	Dynamic	Static	Dynamic	Static
BLAS	libmkl.so libguide.so	libmkl_ipf.a libguide.a[1]	libpthread.so	libpthread.a
Sparse BLAS	libmkl.so libguide.so	libmkl_ipf.a libguide.a	libpthread.so	libpthread.a
BLAS95 Interface	libmkl_blas95.a libmkl.so libguide.so	libmkl_blas95.a libmkl_ipf.a libguide.a	libpthread.so	libpthread.a
CBLAS	libmkl.so libguide.so	libmkl_ipf.a libguide.a	libpthread.so	libpthread.a
LAPACK	libmkl_lapack32.so and/or libmkl_lapack64.so libmkl.so libguide.so	libmkl_lapack.a libmkl_ipf.a libguide.a	libpthread.so	libpthread.a
LAPACK95 Interface	n/a[2]	libmkl_lapack95.a libmkl_lapack.a libmkl_ipf.a libguide.a	libpthread.so	libpthread.a
Sparse Solver	n/a	libmkl_solver.a libmkl_lapack.a libmkl_ipf.a libguide.a	libpthread.so	libpthread.a
Vector Math Library	libvml.so libguide.so	libmkl_ipf.a libguide.a	libpthread.so libm.so	libpthread.a libm.a

Function domain / Interface	Intel MKL libraries in lib/64 directory		Linux libraries	
	Dynamic	Static	Dynamic	Static
Vector Statistical Library	libmkl_lapack32.so and/or libmkl_lapack64.so libmkl.so libvsl.so libguide.so	libmkl_lapack.a libmkl_ipf.a libguide.a	libpthread.so libm.so	libpthread.a libm.a
Fourier Transform Functions	libmkl.so libguide.so	libmkl_ipf.a libguide.a	libpthread.so libm.so (optionally)	libpthread.a libm.a (optionally)
Interval Arithmetic	libmkl_lapack32.so and/or libmkl_lapack64.so libmkl.so libguide.so	libmkl_lapack.a libmkl_ipf.a libguide.a	libpthread.so	libpthread.a
Trigonometric Transform Functions	libmkl.so libvml.so libguide.so	libmkl_ipf.a libguide.a	libpthread.so	libpthread.a
Poisson Library	libmkl.so libvml.so libguide.so	libmkl_ipf.a libguide.a	libpthread.so	libpthread.a

[1] Regardless of the function domain, when linking statically with `libguide` (discouraged), sometimes, you may have to use the libguide version different from the one in the indicated directory (see Notes below).

[2] Not applicable

3.5 Linking Examples

Below are some specific examples for linking on IA-32 systems with Intel® compilers:

`ifort myprog.f -L$MKLPATH -lmkl_lapack -lmkl_ia32 -lguide -lpthread`
static linking of user code `myprog.f`, LAPACK, and kernels. Processor dispatcher will call the appropriate kernel for the system at runtime.

`ifort myprog.f -L$MKLPATH -lmkl_lapack95 -lmkl_lapack -lmkl_ia32 -lguide -lpthread`
static linking of user code `myprog.f`, Fortran-95 LAPACK interface, and kernels. Processor dispatcher will call the appropriate kernel for the system at runtime.

`ifort myprog.f -L$MKLPATH -lmkl_blas95 -lmkl_lapack -lmkl_ia32 -lguide -lpthread`
static linking of user code `myprog.f`, Fortran-95 BLAS interface, and kernels. Processor dispatcher will call the appropriate kernel for the system at runtime.

`icc myprog.c -L$MKLPATH -lmkl_ia32 -lguide -lpthread -lm`
static linking of user code `myprog.c`, BLAS, Sparse BLAS, GMP, VML/VSL, interval arithmetic, and FFT/DFT. Processor dispatcher will call the appropriate kernel for the system at runtime.

`ifort myprog.f -L$MKLPATH -lmkl_solver -lmkl_lapack -lmkl_ia32 -lguide -lpthread`
static linking of user code `myprog.f`, the sparse solver, and possibly other routines within Intel MKL (including the kernels needed to support the sparse solver).

`icc myprog.c -L$MKLPATH -lmkl -lguide -lpthread`
dynamic linking of user code `myprog.c`, the BLAS or FFTs within Intel MKL.

Notes:

- When using the Intel® MKL shared libraries, do not forget to update the shared libraries environment path, i.e. a system variable LD_LIBRARY_PATH, to include the libraries location. For example, if the Intel MKL libraries are in the `/opt/intel/mkl/8.0/lib/32` directory, then the following command line can be used (assuming a bash shell):

  ```
  export LD_LIBRARY_PATH=/opt/intel/mkl/9.0/lib/32:$LD_LIBRARY_PATH
  ```

- If you link `libguide` statically (discouraged) and use the Intel compilers, then link in the `libguide` version that comes with the compiler, that is, use `-openmp` option.
- If you link `libguide` statically but do not use the Intel compiler, then link in the `libguide` version that comes with Intel MKL.

- If you use dynamic linking (`libguide.so`) of Intel MKL (recommended), make sure the `LD_LIBRARY_PATH` is defined so that exactly this version of `libguide` is found and used at runtime.

For linking examples, see the Intel MKL support website at http://www.intel.com/support/performancetools/libraries/mkl/.

3.6 Using language-specific interfaces with Intel MKL

The following interface libraries and modules may be generated as a result of operation of respective makefiles located in the `interfaces` folder:

File name	Comment
`libmkl_blas95.a`	Contains Fortran-95 wrappers for BLAS (BLAS95)
`libmkl_lapack95.a`	Contains Fortran-95 wrappers for LAPACK (LAPACK95)
`Libfftw2xc_gnu.a`	Contains interfaces for FFTW version 2.x (C interface for GNU compiler) to call Intel MKL DFTI
`Libfftw2xc_intel.a`	Contains interfaces for FFTW version 2.x (C interface for Intel® compiler) to call Intel MKL DFTI
`Libfftw2xf_gnu.a`	Contains interfaces for FFTW version 2.x (Fortran interface for GNU compiler) to call Intel MKL DFTI
`Libfftw2xf_intel.a`	Contains interfaces for FFTW version 2.x (Fortran interface for Intel compiler) to call Intel MKL DFTI
`Libfftw3xc_gnu.a`	Contains interfaces for FFTW version 3.x (C interface for GNU compiler) to call Intel MKL DFTI
`Libfftw3xc_intel.a`	Contains interfaces for FFTW version 3.x (C interface for Intel compiler) to call Intel MKL DFTI
`Libfftw3xf_gnu.a`	Contains interfaces for FFTW version 3.x (Fortran interface for GNU compiler) to call Intel MKL DFTI
`Libfftw3xf_intel.a`	Contains interfaces for FFTW version 3.x (Fortran interface for Intel compiler) to call Intel MKL DFTI
`mkl95_blas.mod`	Contains Fortran-95 interface module for BLAS (BLAS95)
`mkl95_lapack.mod`	Contains Fortran-95 interface module for LAPACK (LAPACK95)
`mkl95_precision.mod`	Contains Fortran-95 definition of precision parameters for BLAS95 and LAPACK95.

Section "Fortran-95 interfaces and wrappers to LAPACK and BLAS" shows by example how these libraries and modules are generated.

Fortran-95 interfaces and wrappers to LAPACK and BLAS

Fortran-95 interfaces and wrappers are delivered as sources. The simplest way to use them is building corresponding libraries and linking them as user's libraries. To do this, you must have administrator rights. Provided the product directory is open for writing, the procedure is simple:

Go to the respective directory `mkl/9.0_beta/interfaces/blas95` or `mkl/9.0_beta/interfaces/lapack95`
and type one of the following commands:

```
make PLAT=lnx32 lib     - for IA-32
make PLAT=lnx32e lib    - for Intel® EM64T
make PLAT=lnx64 lib     - for Intel® Itanium® 2 processor platform.
```

As a result, the required library and a respective .mod file will be built and installed in the standard catalog of the release. The .mod files can also be obtained from files of interfaces, using the compiler command
`ifort -c mkl_lapack.f90` or `ifort -c mkl_blas.f90`.
These files are in the include directory.
If you do not have administrator rights, do the following:

1. copy the entire directory (`mkl/9.0_beta/interfaces/blas95` or `mkl/9.0_beta/interfaces/lapack95`) into a user-defined directory <*user_dir*>
2. copy the corresponding file (`mkl_blas.f90` or `mkl_lapack.f90`) from `mkl/9.0_beta/include` into the user-defined directory <*user_dir*>`/blas95` or <*user_dir*>`/lapack95` respectively
3. run one of the above commands in <*user_dir*>`/blas95` or <*user_dir*>`/lapack95` with an additional variable, for instance,
   `make PLAT=lnx32 INTERFACE=mkl_blas.f90 lib`
   `make PLAT=lnx32 INTERFACE=mkl_lapack.f90 lib`.

Now the required library and the `.mod` file will be built and installed in the <*user_dir*>`/blas95` or <*user_dir*>`/lapack95` directory, respectively.

By default, the ifort compiler is assumed. You may change it with an additional parameter of make: FC=<compiler>.
For instance,

```
make PLAT=lnx64 FC=<compiler> lib
```

There is also a way to use the interfaces without building the libraries.

To delete library from the building directory, use the following commands:

```
make PLAT=lnx32 clean        - for IA-32
make PLAT=lnx32e clean       - for Intel® EM64T
make PLAT=lnx64 clean        - for Intel® Itanium® 2 processor platform.
```

3.7 Building custom shared objects

Custom shared objects enable reducing the collection of functions available in Intel MKL libraries to those required to solve your particular problems, which helps to save disk space and build your own dynamic libraries for distribution.

Intel MKL custom shared object builder

Custom shared object builder is targeted for creation of a dynamic library (shared object) with selected functions and located in `tools/builder` folder. The builder contains a makefile and a definition file with the list of functions. The makefile has three targets: "ia32", "ipf", and "em64t". ia32 target is used for IA-32, ipf is used for Intel® Itanium® processor family and em64t is used for Intel® Xeon® processor with Intel® EM64T.

Specifying makefile parameters

There are several macros (parameters) for the makefile:

`export = `*`functions_list`*

determines the name of the file that contains the list of entry point functions that will be included into shared object. This file is used for definition file creation and then for export table creation. Default name is `functions_list`.

`name = `*`mkl_custom`*

specifies the name of the created library. By default, the library `mkl_custom.so` is built.

`xerbla = `*`user_xerbla.obj`*

specifies the name of object file that contains user's error handler. This error handler will be added to the library and then will be used instead of standard MKL error handler `xerbla`. By default, that is, when this parameter is not specified, standard MKL `xerbla` is used.

All parameters are not mandatory. For the simplest case, the command line could be `make ia32` and the values of the remaining parameters will be taken by default. As a

result, `mkl_custom.so` library for IA-32 will be created, the functions list will be taken from `functions_list` file, and the standard MKL error handler `xerbla` will be used.

Another example for a more complex case is as follows:
```
make ia32 export=my_func_list.txt name=mkl_small
xerbla=my_xerbla.o
```
In this case, `mkl_small.so` library for IA-32 will be created, the functions list will be taken from `my_func_list.txt` file, user's error handler `my_xerbla.o` will be used.

The process is similar for the Intel® Itanium® processor family applications and Intel® EM64T applications.

Specifying list of functions

Entry points in `functions_list` file should be adjusted to interface. For example, Fortran functions get an underscore character "_" as a suffix when added to the library:

```
dgemm_
ddot_
dgetrf_
```

If selected functions have several processor-specific versions, they all will be included into the custom library and managed by dispatcher.

4 Managing Performance and Memory

The chapter discusses ways to obtain best performance with Intel MKL as well as Intel MKL memory management.

4.1 Performance

The section discusses Intel MKL parallelism and ways to avoid conflicts in the threaded execution environment; shows how to set up the number of threads and change it during run time; gives recommendations on multi-core performance. The section also shows other ways to gain performance, for example, by proper data alignment.

4.1.1 Using Intel MKL parallelism

Intel® MKL is threaded in a number of places: direct sparse solver, LAPACK (`*GETRF`, `*POTRF`, `*GBTRF`, `*GEQRF`, `*ORMQR`, `*STEQR`, `*BDSQR` routines), all Level 3 BLAS, Sparse BLAS matrix-vector and matrix-matrix multiply routines for the compressed sparse row and diagonal formats, and all DFTs (except 1D transformations when `DFTI_NUMBER_OF_TRANSFORMS=1` and sizes are not power of two).

There are situations in which conflicts can exist in the execution environment that make the use of threads in Intel® MKL problematic. They are listed here with recommendations for dealing with these. First, a brief discussion of why the problem exists is appropriate.

If the user threads the program using OpenMP* directives and uses the Intel compilers to compile the program, Intel® MKL and the user program will both use the same threading library. Intel® MKL tries to determine if it is in a parallel region in the program, and if it is, it does not spread its operations over multiple threads. But Intel® MKL can be aware that it is in a parallel region only if the threaded program and Intel® MKL are using the same threading library. If the user program is threaded by some other means, Intel® MKL may operate in multithreaded mode and the computations may be corrupted.

Here are several cases with recommendations depending on the threading model you employ:

Table 7 How to avoid conflicts in the computation environment for your threading model

Threading model	Discussion
You thread the program using OS threads (`pthreads` on Linux*).	If more than one thread calls the library, and the function being called is threaded, it is important that you turn off Intel® MKL threading. Set `OMP_NUM_THREADS=1` in the environment. This is the default with Intel® MKL except for the Direct Sparse Solver.
You thread the program using OpenMP* directives and/or pragmas and compiles the program using a compiler other than a compiler from Intel.	This is more problematic in that setting `OMP_NUM_THREADS` in the environment affects both the compiler's threading library and the threading library with Intel® MKL. At this time, a safer approach is to set `MKL_SERIAL=YES` (or `MKL_SERIAL=yes`) which forces Intel® MKL to serial mode regardless of `OMP_NUM_THREADS` value. You can also obtain through your Intel® Premier Support account a sequential version of MKL, containing no threading, which is the safest approach for this case.
There are multiple programs running on a multiple-cpu system, as in the case of a parallelized program running using MPI for communication in which each processor is treated as a node.	The threading software will see multiple processors on the system even though each processor has a separate process running on it. In this case `OMP_NUM_THREADS` should be set to 1. Again, however, the default behavior of Intel MKL is to run with one thread.

Setting up the number of threads

The OpenMP* software responds to the environmental variable `OMP_NUM_THREADS`. The number of threads can be set in the shell the program is running in. To change the number of threads, in the command shell in which the program is going to run, enter:

`export OMP_NUM_THREADS=<number of threads to use>`.

To force the library to serial mode, environment variable `MKL_SERIAL` should be set to `YES`. It works regardless of `OMP_NUM_THREADS` value.

If the variable `OMP_NUM_THREADS` is not set, Intel® MKL software will run on the number of threads equal to 1. It is recommended that you always set `OMP_NUM_THREADS` to the number of processors you wish to use in your application.

NOTE: Currently the default number of threads for Sparse Solver is the number of processors in the system.

Changing the number of processors for threading during run time

It is not possible to change the number of processors during run time using the environment variable `OMP_NUM_THREADS`. However, you can call OpenMP API functions from your program to change the number of threads during runtime. The following

sample code demonstrates changing the number of threads during runtime using the `omp_set_num_threads()` routine:

```
#include "omp.h"
#include "mkl.h"
#include <stdio.h>

#define SIZE 1000

void main(int args, char *argv[]){

   double *a, *b, *c;
   a = new double [SIZE*SIZE];
   b = new double [SIZE*SIZE];
   c = new double [SIZE*SIZE];

   double alpha=1, beta=1;
   int m=SIZE, n=SIZE, k=SIZE, lda=SIZE, ldb=SIZE, ldc=SIZE, i=0,
j=0;
   char transa='n', transb='n';

   for( i=0; i<SIZE; i++){
       for( j=0; j<SIZE; j++){
           a[i*SIZE+j]= (double)(i+j);
           b[i*SIZE+j]= (double)(i*j);
           c[i*SIZE+j]= (double)0;
       }
   }
   cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
                  m, n, k, alpha, a, lda, b, ldb, beta, c, ldc);

   printf("row\ta\tc\n");
   for ( i=0;i<10;i++){
       printf("%d:\t%f\t%f\n", i, a[i*SIZE], c[i*SIZE]);
   }

   omp_set_num_threads(1);

   for( i=0; i<SIZE; i++){
       for( j=0; j<SIZE; j++){
           a[i*SIZE+j]= (double)(i+j);
           b[i*SIZE+j]= (double)(i*j);
           c[i*SIZE+j]= (double)0;
       }
   }
   cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
                  m, n, k, alpha, a, lda, b, ldb, beta, c, ldc);

    printf("row\ta\tc\n");
    for ( i=0;i<10;i++){
        printf("%d:\t%f\t%f\n", i, a[i*SIZE],
c[i*SIZE]);
    }

    omp_set_num_threads(2);
    for( i=0; i<SIZE; i++){
```

```
        for( j=0; j<SIZE; j++){
            a[i*SIZE+j]= (double)(i+j);
            b[i*SIZE+j]= (double)(i*j);
            c[i*SIZE+j]= (double)0;
        }
    }
    cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
                  m, n, k, alpha, a, lda, b, ldb, beta, c, ldc);

    printf("row\ta\tc\n");
    for ( i=0;i<10;i++){
        printf("%d:\t%f\t%f\n", i, a[i*SIZE],
c[i*SIZE]);
    }

    delete [] a;
    delete [] b;
    delete [] c;
```

4.1.2 Multi-core performance

You may experience the issues that the application using Intel MKL runs faster when threaded on the number of sockets rather than on the number of cores, and/or parallel application performance is unstable on multi-core. Binding threads to the CPU cores may improve or stabilize the performance. This is performed by setting an affinity mask to threads. You can do it either with OpenMP facilities (which is recommended if available, for instance, via `KMP_AFFINITY` environment variable using Intel OpenMP), or with a system routine (see the example below).

Example of setting an affinity mask by operating system means using Intel compiler

Suppose, the system has two sockets with two cores each, and 2 threads parallel application, which calls Intel MKL DFT, happens to run faster than in 4 threads, but the performance in 2 threads is very unstable. Put the fragment marked green into your code before DFT call to bind the threads to the cores on different sockets.

```
// Set affinity mask
#include <sched.h>
#include <omp.h>
#pragma omp parallel default(shared) private(mask)
{
   unsigned long mask = (1 << omp_get_thread_num()) * 2;
   sched_setaffinity( 0, sizeof(mask), &mask );
}
// Call MKL DFT routine
...
```

Then build your application and run it in 2 threads:
`env OMP_NUM_THREADS=2 ./a.out`

4.1.3 Setting up data for better performance

To obtain best performance, you should properly align data arrays in your code. The section lists general conditions as well as FFT domain-specific ones. Additionally, in certain cases, the proper matrix format is recommended for LAPACK routines.

General recommendations on data alignment

To obtain the best performance with Intel® MKL, make sure the following conditions are met:

- arrays are aligned on a 16-byte boundary
- leading dimension values (`n*element_size`) of two-dimensional arrays are divisible by 16
- for two-dimensional arrays, leading dimension values divisible by 2048 are avoided.

LAPACK packed routines performance

The routines with the names that contain the letters HP, OP, PP, SP, TP, UP in the matrix type and storage position (the second and third letters respectively) operate on the matrices in the packed format (see "LAPACK Routine Naming Conventions" sections in the Intel MKL Reference Manual). Their functionality is strictly equivalent to the functionality of the unpacked routines with the names containing the letters HE, OR, PO, SY, TR, UN in the corresponding positions, but the performance is significantly lower.
If the memory restriction is not too tight, use an unpacked routine for better performance. Note that in such a case, you need to allocate $N^2/2$ more memory than the memory required by a respective packed routine, where N is the problem size (the number of equations).

For example, solving a symmetric eigenproblem with an expert driver can be speeded up through using an unpacked routine:
`call dsyevx(`*`jobz, range, uplo, n, a, lda, vl, vu, il, iu, abstol, m, w, z, ldz, work, lwork, iwork, ifail, info`*`)`,
where *a* is the dimension *lda*-by-*n*, which is at least N^2 elements, instead of
`call dspevx(`*`jobz, range, uplo, n, ap, vl, vu, il, iu, abstol, m,`*

w, z, ldz, work, iwork, ifail, info),
where *ap* is the dimension N*(N+1)/2.

Data alignment for FFT functions

There are additional conditions for the FFT functions:

On IA-32 based applications the addresses of the first elements of arrays and the leading dimension values, in bytes (`n*element_size`), of two-dimensional arrays should be divisible by cache line size (32 bytes for Pentium® III processor, 64 bytes for Pentium® 4 processor, and 128 bytes for Intel® EM64T processor).
On Itanium®-based applications the sufficient conditions are as follows:

- for the C-style FFT, the distance L between arrays that represent real and imaginary parts is not divisible by 64. The best case is when L=k*64 + 16
- leading dimension values, in bytes (n*element_size), of two-dimensional arrays are not power of two.

4.2 Using Intel MKL Memory Management

Intel® MKL has memory management software that controls memory buffers for use by the library functions. When a call is made to certain library functions (such as those in the Level 3 BLAS or DFTs), new buffers are allocated if there are no free ones (marked as free) currently available. These buffers are not deallocated until the program ends. If at some point your program needs to free memory, it may do so with a call to `MKL_FreeBuffers()`. If another call is made to a library function that needs a memory buffer, then the memory manager will again allocate the buffers and they will again remain allocated until either the program ends or the program deallocates the memory. The memory management software is turned on by default.

This behavior by design facilitates achieving better performance. However, some tools might report this behavior as a memory leak. Should you wish, you can release memory in your program through the use of a function made available in Intel MKL or you can force memory releasing after each call by setting an environment variable.

To disable the memory management software using the environment variable, set `MKL_DISABLE_FAST_MM` to any value, which will cause memory to be allocated and freed from call to call. Disabling this feature will negatively impact performance of routines such as the level 3 BLAS, especially for small problem sizes.

Using one of these methods to release memory will not necessarily stop programs from reporting memory leaks, and, in fact, may increase the number of such reports should you make multiple calls to the library thereby requiring new allocations with

each call. Memory not released by one of the methods described will be released by the system when the program ends.

Memory management has a restriction for the number of allocated buffers in each thread. Currently this number is 32. The maximum number of supported threads is 514. To avoid the default restriction, disable memory management.

Replacing Memory Functions

Intel® MKL memory management uses standard C runtime memory functions to allocate or free memory. Since MKL 9.0, you can replace these memory functions by your own memory functions. The `i_malloc.h` header file contains all declarations required for an application developer to replace the memory allocation functions. This header file describes how memory allocation can be replaced in those Intel(R) libraries that support this feature.

5 Configuring Your Development Environment

This chapter explains how to configure your development environment for the use of Intel MKL and especially what features may be customized using the Intel MKL configuration file.

For information on how to set up the environment variables for threading, refer to Setting up the number of threads.

After the installation of Intel MKL is complete, you can use files `mklvars32.sh`, `mklvarsem64t.sh`, and `mklvars64.sh` in the `tools/environment` directory to set the environment variables `INCLUDE` and `LD_LIBRARY_PATH` in the user shell.

5.1 Customizing Intel MKL Using the Configuration File

Intel MKL configuration file will provide the possibilities to customize several features of the Intel MKL, namely:

- redefine names of downloadable dynamic libraries
- turn on/off checking of the input parameters for possible errors
- set Threaded/Non-Threaded operation mode.

The configuration file is `mkl.cfg` file by default. The file contains several variables that can be changed. Below is the example of the configuration file containing all possible variables with default values:

```
//
// Default values for mkl.cfg file
//
// SO names for IA-32
MKL_X87so = mkl_def.so
MKL_SSE1so = mkl_p3.so
MKL_SSE2so = mkl_p4.so
```

```
MKL_SSE3so = mkl_p4p.so
MKL_VML_X87so = mkl_vml_def.so
MKL_VML_SSE1so = mkl_vml_p3.so
MKL_VML_SSE2so = mkl_vml_p4.so
MKL_VML_SSE3so = mkl_vml_p4p.so
// SO names for Intel(R) 64
MKL_EM64TDEFso = mkl_def.so
MKL_EM64TSSE3so = mkl_p4n.so
MKL_VML_EM64TDEFso = mkl_vml_def.so
MKL_VML_EM64TSSE3so = mkl_vml_p4n.so
// SO names for IA-64
MKL_I2Pso = mkl_i2p.so
MKL_VML_I2Pso = mkl_vml_i2p.so
// DLL names for LAPACK libraries
MKL_LAPACK32so = mkl_lapack32.so
MKL_LAPACK64so = mkl_lapack64.so
// Serial or parallel mode
//     YES - single threaded
//     NO - multi threaded
//     OMP - control by OMP_NUM_THREADS
MKL_SERIAL = YES
// Input parameters check
//     ON - checkers are used (default)
//     OFF - checkers are not used
MKL_INPUT_CHECK = ON
```

When any Intel® MKL function is first called, Intel MKL checks to see if the configuration file exists, and if so, it operates with the specified variables. The path to the configuration file is specified by environment variable `MKL_CFG_FILE`. If this variable is not defined, then first the current directory is searched through, and then the directories specified in the PATH environment variable. If the Intel MKL configuration file does not exist, the library operates with default values of variables (standard names of libraries, checkers on, non-threaded operation mode).
If the variable is not specified in the configuration file, or specified incorrectly, the default value is used.

5.2 Redefining Names of Downloadable Dynamic Libraries

Below is an example of the configuration file that only redefines the library names:

```
// SO redefinition
MKL_X87so = matlab_x87.so
MKL_SSE1so = matlab_sse1.so
MKL_SSE2so = matlab_sse2.so
MKL_SSE3so = matlab_sse2.so
MKL_ITPso = matlab_ipt.so
MKL_I2Pso = matlab_i2p.so
```

6 Coding Intel® Math Kernel Library Calls

Though Intel® Math Kernel Library (Intel® MKL) provides support for Fortran and C/C++ programming, not all function domains support both interfaces (see Table Intel MKL Language Support). For example, LAPACK has no C interface.

To enable calling Intel MKL routines in your environment even if the function domain does not support it, mixed-language programming may be used.

See also Using language-specific interfaces with Intel MKL.

6.1 Calling LAPACK, BLAS, and CBLAS Routines from C Language Environments

The Intel MKL is provided in C and Fortran environments. Not all of the Intel MKL sub-libraries support both environments. In order to use these sub-libraries in both environments, you should observe some "rules".

LAPACK

When calling LAPACK routines from C-language programs, make sure that you follow Fortran rules:

- Pass variables by 'address' as opposed to pass by 'value'.
- Be sure to store your data Fortran-style, i.e. data stored in column-major rather than row-major order.

NOTE: In LAPACK, routine names are upper case only (Please see Example1 in section 6.2.)

BLAS

BLAS routines are Fortran-style routines. If you call BLAS routines from a C-language program, you must follow the Fortran-style calling conventions:

- Pass variables by address as opposed to passing by value.
- Be sure to store data Fortran-style, i.e. data stored in column-major rather than row-major order.

NOTE: In the BLAS, Intel MKL provides both upper case and lower case names to the routines (Please see Example1 in section 6.2.)

An alternative to calling BLAS routines from a C-language program is to use the CBLAS interface.

CBLAS

CBLAS routines are provided as the C-style interface to the BLAS routines. Call CBLAS routines using regular C-style calls. When using the CBLAS interface, the header file `mkl.h` will simplify the program development as it specifies enumerated values as well as prototypes of all the functions. The header determines if the program is being compiled with a C++ compiler, and if it is, the included file will be correct for use with C++ compilation.

6.2 How to Call BLAS Functions That Return the Complex Values in C/C++ Code

When handling a call of a complex BLAS function that returns complex values from C, you must be careful. The problem arises because these are Fortran functions and the return values are handled quite differently for the two languages (C and Fortran) for complex values. Because Fortran lets you call functions as though they were subroutines, however, there is a mechanism for returning the complex value correctly when the function is called from a C program. When a Fortran function is called as a subroutine, the return value shows up as the first parameter in the calling sequence. — This feature can be exploited by the C programmer.

The following example, for `cdotc()`, shows how this works. You call the function from Fortran as follows: `result = cdotc( n, x, 1, y, 1 )`

A call to this function as a subroutine, looks like this: `call cdotc( result, n, x, 1, y, 1)`.

From C, this would look in this way: `cdotc( &result, &n, x, &one, y, &one )` where the hidden parameter is exposed.

NOTE: Intel® MKL has both upper-case and lower-case entry points in the BLAS, so all upper-case or all lower-case names are acceptable.

Using this form, you can call from C, and thus, from C++, several level 1 BLAS functions that return complex values. However, it is still easier to use the CBLAS interface. For instance, you can call the same function using the CBLAS interface as follows:

```
cblas_cdotu( n, x, 1, y, 1, &result )
```

NOTE: The complex value comes back expressly in this case.

***Example 1*: Calling a Complex BLAS Level 1 Function from C**

```
/* The following example illustrates a call from a C program to
the complex BLAS Level 1 function zdotc(). This function computes
the dot product of two double-precision complex vectors.

In this example, the complex dot product is returned in the
structure c.
*/
#include "mkl.h"
#define N 5
void main()
{
int n, inca = 1, incb = 1, i;
typedef struct{ double re; double im; } complex16;
complex16 a[N], b[N], c;
void zdotc();
n = N;
for( i = 0; i < n; i++ ){
a[i].re = (double)i; a[i].im = (double)i * 2.0;
b[i].re = (double)(n - i); b[i].im = (double)i * 2.0;
}
zdotc( &c, &n, a, &inca, b, &incb );
printf( "The complex dot product is: ( %6.2f, %6.2f)\n", c.re,
c.im );
}
```

***Example 2:* Calling a Complex BLAS Level 1 Function from C++**

```
#include "mkl.h"
typedef struct{ double re; double im; } complex16;
extern "C" void zdotc (complex16*, int *, complex16 *, int *,
complex16 *, int *);

#define N 5

void main()
{
int n, inca = 1, incb = 1, i;

complex16 a[N], b[N], c;
```

```
n = N;
for( i = 0; i < n; i++ ){
a[i].re = (double)i; a[i].im = (double)i * 2.0;
b[i].re = (double)(n - i); b[i].im = (double)i * 2.0;
}
zdotc(&c, &n, a, &inca, b, &incb );
printf( "The complex dot product is: ( %6.2f, %6.2f)\n", c.re,
c.im );
}
```

Example 3: **Using the CBLAS Interface Instead of Calling BLAS Directly from C Programs**

```
#include "mkl.h"
typedef struct{ double re; double im; } complex16;

extern "C" void cblas_zdotc_sub ( const int , const complex16 *,
    const int , const complex16 *, const int, const complex16*);

#define N 5

void main()
{

int n, inca = 1, incb = 1, i;

complex16 a[N], b[N], c;
n = N;
for( i = 0; i < n; i++ ){
a[i].re = (double)i; a[i].im = (double)i * 2.0;
b[i].re = (double)(n - i); b[i].im = (double)i * 2.0;
}
cblas_zdotc_sub(n, a, inca, b, incb,&c );
printf( "The complex dot product is: ( %6.2f, %6.2f)\n", c.re,
c.im );
}
```

Appendix A Intel® Math Kernel Library Language Support

The following table lists function domains that Intel® Math Kernel Library comprises as well as programming language support for these domains:

Table 8 Intel MKL language support

Function Domain	Fortran-77	Fortran-90/95	C/C++
Basic Linear Algebra Subprograms (BLAS)	+	+	via CBLAS
Sparse BLAS Level 1	+	+	via CBLAS
Sparse BLAS Level 2 and 3	+	+	+
LAPACK routines for solving systems of linear equations	+	+	
LAPACK routines for solving least-squares problems, eigenvalue and singular value problems, and Sylvester's equations	+	+	
Auxiliary and utility LAPACK routines	+		
PARDISO	+		+
Other Direct and Iterative Sparse Solver routines	+	+	+
Vector Mathematical Library (VML) functions	+		+
Vector Statistical Library (VSL) functions	+		+
Fourier Transform functions (DFT)		+	+
Interval Solver routines	+		
Trigonometric Transform routines		+	+
Fast Poisson, Laplace, and Helmholz Solver (Poisson Library) routines		+	+

Appendix B Contents of the doc Directory

The table below shows the contents of the doc directory.

Table 9 Contents of the doc directory

File name	Comment
mklEULA.txt	Intel MKL license
Doc_Index.htm	Index of Intel MKL documentation
fftw2xmkl_notes.htm	FFTW 2.x Interface Support Technical User Notes
fftw3xmkl_notes.htm	FFTW 3.x Interface Support Technical User Notes
fft2dfti.pdf	Intel FFT to DFTI Wrappers Technical User Notes
Getting_Started.htm	Getting Started with Intel MKL (this document)
Install.txt	Installation Guide
Readme.txt	Initial User Information
redist.txt	List of redistributable files
Release_Notes.htm	Release Notes
Release_Notes.txt	Release Notes (text format)
mklman.pdf	Intel MKL Reference Manual
mklman80_j.pdf	Intel MKL Reference Manual in Japanese
mklqref/index.htm	MKL Quick Reference
mklsupport.txt	Information on package number for customer support reference
mkluse.htm	Technical User notes for Intel MKL
vmlnotes.htm	General discussion of VML
vslnotes.pdf	General discussion of VSL
userguide.pdf	This document.

Index

T

U